Methods and Models for Combinatorial Optimization, Lab Exercise Part 1

Alberto Nicoletti

November 2023, University of Padova

1 The problem

The exercise requires to implement a given model for TSP (travelling salesperson problem), considering its practical application: calculate the sequence of holes on a board to be done from a drilling machine such that the total amount of time is minimized.

2 The implementation

The provided file *main.cpp* implements the linear programming model. The structure of the code has been provided by prof. De Giovanni, as well as the *cpxmacro.h* and the *makefile* files. To run the executable file, an instance of the problem has to be provided as argument (example: tsp6.dat).

This is the (simplified) outline of the program:

- Read from file the number of nodes and the weighted adjacency matrix.
- Create the variables (columns in Cplex API) x's and y's. The variable names are stored as *x_i_j* and *y_i_j*.
- Add the constraints (rows in Cplex API).
- Compute the optimal solution and get the time to find it.
- Print the names and values of the variables for the optimal solution. For readability reasons only the variables corresponding the edges that are part of the solution are printed (i.e. variables values different from zero).
- Print the value of the objective function.
- Print the time needed to find the optimal solution. Note that this time considers only the optimization part, not the set up of the problem nor the printing part.

3 The results

The program correctness has been tested manually for graphs of size 4, 5 and 6.
All the following tests have been done in the computers of LabTA in Torre Archimede. Running the program over the same instance of the problem many times gave slightly different time results, so all the times showed here are the average of 3 runs. The instances of the problem have been created using the program *cdata.cpp*. Two variables have been considered to make the tests: number of nodes and the shape of the instances. Three shapes for the instances have been used.

Random graph

Given the number of nodes, the program assigns randomly an integer weight to each edge. The tests showed that the range of values have a big impact on the performances, on the same level of the number of nodes. Finding a solution for a graph with random weights between 1 and 100 can take even 10 times more time than finding a solution for a problem where the range is 1 to 10.

The instances of this problem are stored in the files *tspN_k.dat* (where N is the number of nodes and the weight range is [1,k]).

Results Maximum number of nodes to find a solution with a weight range [1,100] in less than:

0,1s : Max 11 nodes.

1s : Max 50 nodes.

10s : Max 82 nodes.

	Time		
Number of nodes	Range [1,10]	Range [1,100]	Range [1,1000]
10	0.0452s	0.0245s	0.0534s
20	0.1241s	0.1262s	0.0728s
50	0.8227s	0.7352s	2.1114s
80	0.5715s	5.7106s	11.526s
100	0.9091s	24.92s	31.35s
120	2.0217s	$> 60s$	$> 60s$
150	$> 60s$	$> 60s$	$> 60s$

Table 1: Results for random graphs.

Grid graph

The program creates (implicitly) the graph putting the nodes on a grid. Note that the grid doesn't have to be complete (i.e. the last positions of the last row can be empty). The weights are calculated as the euclidean distance between two nodes as if they were two points in the Cartesian plane. The distance between two close nodes is defined from the parameter *unit*. For example, for the graph in Figure 1 (with *unit*=1), the edge (1,2) has weight 1 and the edge (2,4) has weight $\sqrt{2}$.

Slightly worse performances have been obtained on the change of the parameter *unit*, so for the tests it has been set to 1. Different performances have been obtained on the change of the shape of the grid. Generally, an higher number of columns brings to worse results. To be realistic, the tests have been done on grid graph with similar dimensions.

The instances of this problem are stored in the files *tspNg.dat* (where N is the number of nodes).

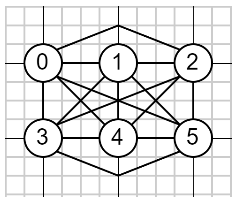

Figure 1: Grid graph of 6 nodes.

Results The change of dimensions in the grid brings an high change in performances. As shown in Table 1, a graph with 80 nodes in a 8x10 grid takes more than 10 times the time it takes for a graph with 80 nodes in a 14x6 grid. This high unpredictability make it difficult to get relevant results. Moreover, considering the gird dimensions as parameters to test would take too much time, having an exponential number of possible instances of the problem.

Number of nodes	Time
70 10 (grid 2x5)	0.0554s
20 (grid 4x5)	0.1309s
50 (grid 10x5)	0.1935s
80 (grid 14x6)	1.9376s
80 (grid 8x10)	28.16s
100 (grid 10x10)	$> 60s$

Table 2: Results for grid graphs.

Random grid graph

This is the case where the shape of the instances is the **closest to the real case** of the boards to be drilled, because of the (assumed) regularity of the position of the holes to be made on the boards.

The program creates (implicitly) a grid and randomly places the nodes on points of that grid. The weights are calculated in the same way as for the grid graph. For example, for the graph in Figure 2 (with *unit*=1), the edge (5,4) has weight 3 and the edge (0,3) has weight $\sqrt{5}$.

The instances of this problem are stored in the files *tspNrg.dat* (where N is the number of nodes).

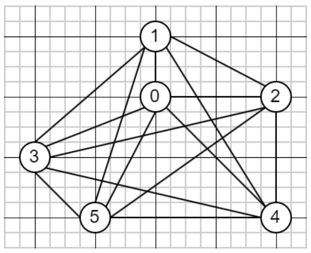

Figure 2: Random grid graph of 6 nodes.

Results Similarly to the case of the grid graph, also in the random grid graph the dimensions of the grid create an high level of uncertainty. It is not possible to provide relevant results about up to which size of the graph the solution is found efficiently. See the different grids for a 50 nodes graph in Table 3 for an example.

Number of nodes	Time
10 (grid 8x10)	0.0563s
20 (grid 8x10)	0.1513s
50 (grid 12x10)	4.233s
50 (grid 6x14)	2.084s
50 (grid 100x5)	$> 60s$
80 (grid 12x10)	28.544s
100 (grid 14x18)	$> 60s$

Table 3: Results for random grid graphs.

Conclusion

For general cases with random graphs the program starts to be inefficient after 50 nodes. The grid graphs and the random grid graphs, more close to our practical application of the problem, show how there is little, if none, guarantee on the efficiency of the program, if we only consider the number of nodes.
The size and dimensions of the grid can change in a significant way for each type of board needed to be produced, so this program (and the relative model) shouldn't be considered an efficient way to find the best sequence of holes to minimize the drilling time.